# Empezando con Spring Framework

E

Elias Ribeiro

29/10/2020

Se nos pidió implementar un sistema en una tienda de productos electrónicos que ayudará a controlar las cantidades de productos que tenemos en la tienda.

Comenzamos alineando las operaciones que el cliente esperaba poder realizar en este control de productos.

Según él, primero era necesario **insertar un producto**, enseguida necesitaban poder siempre **ver cuántos productos había**.

Entonces comenzamos desarrollando este código para agregar un producto.

```
public void agregarProducto(Producto producto) {
        try {
            String sql = "insert into productos (nombre,cantidad) values (?,?)
            PreparedStatement stmt = connection.prepareStatement(sql);
            stmt.setString(1, producto.getNombre());
            stmt.setString(2, producto.getCantidad());
            stmt.execute();
            stmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
```

```
        }
    }
```

En este caso, usamos **Servlets**, pero nos dimos cuenta de que el código resultó ser muy grande y ni siquiera hicimos la mitad de las cosas que pidió el cliente. Solo solucionamos la parte de inserción de productos, pero aún faltan las features para ver, actualizar y eliminar.

Si tenemos en cuenta eso, al final tendríamos un código enorme y, en consecuencia, perjudicaríamos la legibilidad y el mantenimiento.

¿Y cómo podríamos solucionar este problema?

## Usando Spring Framework

Una forma más elegante de solucionar este problema de códigos largos y poco legibles puede ser utilizando Spring.

Spring es un framework que facilita mucho el desarrollo, es decir, usándolo tenemos una mayor legibilidad de código, implementaciones con menos código, facilidad en el mantenimiento de código. Con eso, también tendremos una **entrega más rápida**.

Para comenzar, primero necesitamos crear un proyecto.

## Creando el proyecto

Para crear nuestro proyecto tenemos que acceder al sitio web de spring y colocar las dependencias que queramos, pero ¿cuáles queremos?

Nuestra aplicación tiene que ser **web** para que sea un acceso más dinámico, es decir, podemos acceder desde cualquier dispositivo.

Como cualquier aplicación web, tiene datos, y estos **datos** se deben guardar en algún lugar, por lo que necesitamos una **base de datos**.

Tienes que mapear las clases al banco para acelerar el proceso en relación al banco.

Entonces usemos la dependencia **Web** para que sea una aplicación web que contenga tomcat ya incluido para que no tengamos que hacer toda la configuración.

**Mysql** será la base de datos que usaremos en esta aplicación, porque tengo que guardar los datos en alguna parte. Con esto, además de guardar, podemos cambiar, eliminar o buscar datos que se encuentran dentro de la base.

**JPA** para que persistan nuestras clases en la base de datos, de modo que mapearemos nuestras clases y las crearemos automáticamente.

Vamos a nombrar al **grupo** como .com.sistema para que sea el grupo de nuestra aplicación y definiremos **artifact** como usuarios.

Hecho esto generamos el proyecto haciendo click en **Generate Project** y se generará un zip, pero ¿qué debemos hacer con él? Necesitamos extraerlo para tomar el proyecto que acabamos de crear e importarlo a algún lugar. Lo importaremos a IDE.

## Configurando el proyecto

Con esta importación ya realizada, ejecutaremos el proyecto para ver si todo está bien. Pero al ejecutarlo tuvimos un error.

¿Por qué ocurre este error, si configuramos nuestro proyecto correctamente? Tenga en cuenta que, el error se produjo al configurar el DataSource, ya que faltaron algunos datos.

Sucede porque cuando ponemos la dependencia del **Mysql** tenemos que configurarlo en nuestra aplicación. Pero, ¿dónde deberíamos realizar esta configuración?

Tenemos que hacerla en **application.properties**, aquí es donde se encuentran todas las configuraciones de propiedades de la aplicación.

Ahora tomemos un ejemplo similar al que se encuentra en la documentación del spring. Lo primero que se debe hacer es pasar la url.

En nuestra url pasaremos el tipo de la base, que es mysql, y el puerto donde se ejecuta la base, y haremos un parámetro llamado **createDatabaseIfNotExist** como true, de modo que cree la base si no existiera.

También tenemos el tema de los horarios, así que dejemos el serverTimezone cómo UTC para que no ocurra ningún error.

Hasta ahora tenemos el archivo **application.properties** así:

```
spring.datasource.url = jdbc: mysql:// localhost: 3306 / productos? createData
```

Iniciemos el proyecto para ver si todo está bien. Bueno, al ejecutarlo tendremos otro error, este error nos dice que se nos niega el permiso.

Todavía tenemos que pasar el nombre de usuario y la contraseña para iniciar sesión, de modo que tengamos acceso a la base, sin eso siempre se denegará el acceso. ¿Cómo pasamos el nombre de usuario y la contraseña?

Tendremos que editar el **application.properties** y pasar nuestro nombre de usuario y contraseña de la base de datos. En mi caso, no tengo contraseña en la base, así que dejo el campo vacío.

```
spring.datasource.username = elias
spring.datasource.password =
```

Además de seguir actualizando nuestras entidades tenemos que definir la **propiedad del ddl-auto**. Bueno, vamos a establecer la propiedad de update, de modo que cada vez que se actualicen nuestras entidades, la base también se actualice.

```
spring.jpa.hibernate.ddl-auto = actualizar
```

Pero, ¿qué son estas entidades?

## Entidades

Entidades son modelos de nuestra tabla que estarán en la **base de datos**. Pero, ¿cómo un modelo será una tabla en nuestra base de datos? Cuando configuramos el **hibernate** en nuestra aplicación, el hace la persistencia de los datos en la base de datos, es decir, los atributos que creamos en la clase.

Sin embargo, ¿cómo es posible saber que una **clase es una entidad**? Sabemos que en Java hay anotaciones, entonces busquemos una anotación que diga que la clase es una entidad. Cuando abrimos el enlace de documentación, inmediatamente miramos la anotación **@Entity**.

¿Cómo funciona el **@Entity**? Él mapea la clase para la base, pero cuando hacemos esta anotación debemos mapear el **ID** de la clase. Pero, ¿cómo se hace este **ID**? Para generar el id pasamos otra anotación llamada **@Id**. Pero, ¿cómo se hace este Id? ¿Uno en uno, dos en dos?

Para saber la cantidad de incremento que vamos a pasar, tendremos que usar otra anotación llamada **@GeneraredValue** que nos generará un valor, pero esto aún no soluciona nuestro problema, porque no sabemos cuánto incrementa, para solucionar esto, usaremos el parámetro **(strategy= GenerationType.IDENTITY)** que aumenta en uno, por lo que la identificación no se repite.

Sabiendo todo esto, necesitamos algo para mapear los atributos de **Nombre** y **Cantidad**, como nombre y cantidad están vinculados a un producto, entonces creemos un paquete llamado modelo. Dentro de este paquete también crearemos una clase producto con estos atributos y mapearemos como una entidad.

```
@Entidad
público clase Producto {
    @Carné de identidad
    @GeneratedValue(estrategia = GenerationType.IDENTITY)
    privado Long id;
    privado String Nombre;
    privado doble cantidad;

// getters omitidos
}
```

Ahora hemos creado estos atributos en la base de datos, pero aún no hemos insertado, buscado o cambiado nada hasta ahora. ¿Y cómo se puede hacer esto? ¿Es posible hacerlo directamente en la Entidad?

## Repositorio

Cuando pensamos en insertar, buscar, cambiar datos, inmediatamente pensamos en **DAOS** o incluso en comandos SQL, esto es mucho trabajo. Pero entonces, ¿cómo haremos la persistencia de los datos? ¿En la clase entidad?

Haremos esto en los repositorios, porque facilita el proceso de **CRUD**. Por buenas prácticas no lo haremos dentro de la clase Entidad.

El repositorio tiene una interfaz llamada CrudRepository que nos permite hacer un CRUD de nuestros datos, sin que tengamos que escribir ninguna línea de código.

Necesitamos crear un paquete llamado repositorio y una interfaz llamada ProdutosRepository.

En esta interfaz, ¿qué necesitamos para hacer con que ella haga esta persistencia de datos? Tenemos que extender la interfaz **CrudRepository** y pasar la entidad que queremos y su tipo de ID. Con esto, nuestro **CRUD** está listo.

```
público interfaz ProductosRepositorio extiende CrudRepository<Producto, Largo>
```

Pero, ¿cómo controlamos la aplicación para saber qué vamos a hacer?

## Control

Los controles son el intermediario de nuestra aplicación, es decir, hacen la comunicación con la parte frontal de nuestra aplicación y con la base de datos. Luego, pueden hacer inserciones en la base de datos. Controlamos la aplicación según el protocolo http. ¿Cómo podemos ver el **http tiene los métodos: POST, GET**. ¿Los vamos a utilizar para controlar nuestra aplicación?

Haremos este control a través de anotaciones, mapeando con **@GetMapping, @PostMapping** para seguir el protocolo http. Bien, ahora sabemos para qué sirven estas anotaciones y el protocolo http. Pero, ¿cómo podemos hacer esto en la aplicación?

Bueno, podemos crear un paquete llamado **controller** y una clase llamada ProductoController para controlar el flujo de nuestra aplicación. Pero, si usamos la anotación @Entity para decir que la clase Producto es una entidad, ¿tenemos que usar alguna anotación para el control?

Podemos hacer dos anotaciones **@Controller y el @RestController**. El @Controller se usa para señalar que es una clase de **Spring MVC** y se usa ampliamente para redirigir views. Ya el @RestController hace automáticamente todo lo que @Controller hace, devuelve todo en

JSON y no necesitamos usar la anotación **@ResponseBody**, porque ya está contenido en él, así que usemos @RestController.

Tenemos que mapear nuestra clase con @RestController quedando de esta manera.

```
@RestController
público clase ProductController{

}
```

Si ejecutamos el proyecto, todavía no pasa nada, ya que no hemos mapeado la URL del controller, ¿cómo debemos hacer esto?

Si pensó que una anotación era correcta, tenemos que usar una anotación para hacer esto, esta anotación será @RequestMapping ("url que queremos").

En nuestro caso lo haremos en la url (/ api / producto) siendo el siguiente:

```
@RequestMapping("/ api / product /")
@RestController
público clase ProductController{

}
```

Hacerlo de esta manera todavía no nos devolverá nada, ya que no hemos realizado ningún tipo de mapeo de retorno. ¿Cómo hacer este mapeo? Usando http a nuestra ventaja y haciendo el mapeo con las anotaciones @GetMapping, @PostMapping.

```
@RequestMapping("/ api / product /")
@RestController
público clase ProductController{
    @GetMapping()
    @PostMapping()
}
```

Pero al igual que con el RequestMapping tenemos que pasar una ruta para cada anotación, solo que tienen algo diferente, es decir, en lugar de pasarlo directamente, tenemos que usar el parámetro **value** y su mapeo.

```
@RequestMapping("/ api / product /")
@RestController
público clase ProductController{
    @GetMapping(valor = "lista de productos")
    @PostMapping(valor = "insertar productos")

}
```

Hecho esto nos damos cuenta de que tenemos errores que ocurren debido a que no tenemos nada para hacer uso de estas anotaciones, ¿qué podemos hacer para usarlas?

Una buena idea sería usar métodos con retornos para que sea más fácil de usar, pues si tenemos que listar usuarios es crear un método que contenga un retorno de una lista de usuarios.

```
@GetMapping(valor = "lista de productos")
público Lista <Producto> lista de productos() {

}
```

¿Pero qué vamos a devolver? Pues recuerda nuestro repositorio, allí tenemos muchos métodos y uno de ellos es findAll que busca todos los elementos que hay en la base. Tenemos que hacer la inyección de dependencia del repositorio, podemos hacer esta inyección por el constructor, pero elegiremos hacerlo con una anotación llamada **@Autowired** y utilizar los métodos del repositorio.

```
@Autowired
privado ProductRepository productRepository;
```

Bueno, probemos esto, usemos el **método findAll** para ver si nos devuelve los registros de la base de datos.

```
@GetMapping(valor = "lista de productos")
público Lista <Produto> lista de productos() {
```

```
    regreso productRepositorio.findAll ();
 }
```

Bueno, pero cuando vayamos a probar queremos que regresemos un **JSON**, pero, ¿cómo se puede hacer esto?

Está el `produces` que es el valor que vamos a producir y el `consumes` que es el valor que vamos a consumir. Para nuestro GetMapping produciremos el valor en JSON, de esta manera.

```
@GetMapping(valor = "listProducts", produce = MediaType.APPLICATION_JSON_VALUE
público Lista <Produto> lista de productos() {
    regreso  (Lista <Produto> ) productRepositorio.findAll ();
 }
```

Ahora que sabemos esto en nuestro @PostMapping, ¿cómo sería? Con el PostMapping sería diferente, porque en él tendríamos que consumir los datos de la parte frontal para insertarlo en la base de datos, pero para poder enviar estos datos desde la parte frontal de la aplicación a su control, ¿qué tenemos que hacer?

Tenemos una anotación llamada @RequestBody para enviar los datos por el cuerpo de la requisición, y la usaremos.

Pero, ¿qué tipo de método creo para eso? Bueno, creemos una ResponseEntity para devolver las respuestas como http status code que es una de las mejores prácticas que debemos seguir.

```
@PostMapping(valor = "insertProducts", consumes = MediaType.APPLICATION_JSON_V
público ResponseEntity Anadir Producto(Producto @RequestBody Product) {
    productRepository.save (usuario);
regreso ResponseEntity.status (201) .construir ();
}
```

## Conclusión

Tuvimos el problema en relación a los **DAOS** de ser muy largos y decidimos usar Spring de otra manera.

Cuando usamos Spring tenemos las ventajas del framework y logramos hacer un **CRUD** con rapidez.

Entonces, ¿qué te pareció la publicación? Ahora es incluso más fácil trabajar con Spring, ¿verdad? Si quieres seguir estudiando sobre el tema, ¡mira nuestros cursos de Java en Alura!

ARTÍCULOS DE TECNOLOGÍA > PROGRAMACIÓN

- Foro y comunidad exclusiva para resolver tus dudas
- Acceso a todo el contenido de la plataforma por 6 meses

**¡QUIERO EMPEZAR A ESTUDIAR!**

Paga en moneda local en los siguientes países

**ANUAL**

# US$79,90

un solo pago de US$79,90

- 218 cursos
- Videos y actividades 100% en Español
- Certificado de participación
- Estudia las 24 horas, los 7 días de la semana
- Foro y comunidad exclusiva para resolver tus dudas

Nuevos cursos
cada semana

**NAVEGACIÓN**

PLANES
INSTRUCTORES
BLOG
POLÍTICA DE PRIVACIDAD
TÉRMINOS DE USO
SOBRE NOSOTROS
PREGUNTAS FRECUENTES

**¡CONTÁCTANOS!**

¡QUIERO ENTRAR EN CONTACTO!

**BLOG**

PROGRAMACIÓN
FRONT END
DATA SCIENCE
INNOVACIÓN Y GESTIÓN
DEVOPS

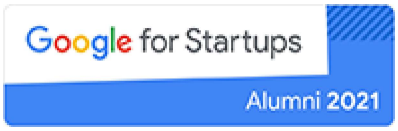
AOVS Sistemas de Informática S.A
CNPJ 05.555.382/0001-33

**SÍGUENOS EN NUESTRAS REDES SOCIALES**

**ALIADOS**

Empresa participante do SCALE UP ENDEAVOR

En Alura somos unas de las Scale-Ups seleccionadas por Endeavor, programa de aceleración de las empresas que más crecen en el país.




Fuimos unas de las 7 startups seleccionadas por Google For Startups en participar del programa Growth Academy en 2021

POWERED BY

## CURSOS

**Cursos de Programación**
Lógica de Programación | Java

**Cursos de Front End**
HTML y CSS | JavaScript | React

**Cursos de Data Science**
Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

**Cursos de DevOps**
Docker | Linux

**Cursos de Innovación y Gestión**
Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento